Applying compress techniques to lower latency on real time object detection application in Android device.

Duc Duc Tran
d.d.tran@student.utwente.nl
University of Twente
Enschede, The Netherlands

ABSTRACT

Deep neural networks have achieved promising results in object detection tasks. However, state-of-the-art networks are computationally expensive due to thousands of parameters, making them not efficient to deploy on hardware-constrained systems such as mobile phones or edge devices. To this end, model compression approaches like pruning and quantization have shown promising improvements to reduce models' complexity with low-performance costs. This work will address the possibilities to apply those model compression techniques to object detection models, enabling the models to work on edge devices. In this work, we first explore the state-of-the-art object detection model MobileNetv2-SSD, then use low-magnitude pruning to remove the redundant parameters in the model. We will further convert this model to TensorFlow Lite format with post-training quantization and deploy it to Android devices to evaluate the latency and accuracy of the new model on object detection tasks. The final model runs on Google Glass Enterprise Edition 2 in 10+ FPS with 72% of parameters pruned and the model is integer quantized without significant loss in accuracy.

KEYWORDS

prunning, quantization, object detection, compressed objects detection

1 INTRODUCTION

The computationally expensive nature of deep neural networks poses challenges for object detection applications, particularly in resource-constrained environments, necessitating the need to reduce costs and improve sustainability. Object detection applications are often cooperated with the usage of complex deep neural networks. With the rise of IoT and edge devices, more and more deep neural networks are incorporated into those systems. However, deep neural networks are powerful yet very computationally expensive. Due to the fact that deep neural networks have several layers with millions of parameters, they usually give good performance. However, they need tons of training data and needs to perform billions of arithmetic operations, thus making the deeps neural networks rely heavily on high-performance hardware such as GPUs, while real-world applications are usually restricted in hardware resources (e.g. mobile phones and embedded devices) [28]. Computationally expensive also means reduction in the sustainability of the system. OpenAI's GPT-3, according to the research paper of Hugging Face, emitted around 550 tons of CO2 for its 14.8 days of training [27]. Thus, it is important to reduce the expensiveness of the networks to first allow the models to run on hardware-constrained devices and second to improve the sustainability of the system.

Numerous studies have been conducted on reducing resource consumption in object detection models through compression techniques like pruning and quantization ([6], [12], [24], [15]). Compression techniques have shown to be a promising way to reduce DNNs' memory and latency with little loss in accuracy. Pruning reduces the complexity of the model by removing non-necessary elements in different layers ([23], [26]). Quantization converts 32-bit floating point weights to a lower resolution 8-bit integer thus can reduce computational and memory costs ([11], [17]). The experimental results of those techniques mentioned on [6] suggest that the models can be compressed with little loss in accuracy while reducing millions of parameters, thus a lot of computation power and memory resources saved. However, many of the researches run and evaluate compressed models on PCs, which have way more hardware resources than an edge device.

Thus, my research will try to address whether a compressed model can run on hardware-restricted devices such as mobile phones and edge devices with reasonable performances. This work will first discover the possibilities of applying low-magnitude pruning over the MobileNetv2-SSD object detection model. Then the model(s) will be converted to TensorFlow lite format with post-training quantization and then deployed to Samsung Galaxy S23 and Google Glass Enterprise Edition 2 to evaluate the performance in terms of frames per second.

The research thus contributes to guiding engineers towards choosing the suitable model(s) and compress techniques for their application noticing the trade-offs between different models and techniques. It also proposes an overview of the performance of pruned and quantized MobileNetv2-SSD on Google Glass Enterprise Edition 2, which has limited studies at the time this work was conducted.

The rest of the paper will first go through the general knowledge of object detection models and pruning on object detection models in section 2. Section 3 discusses in detail our model in this work MobileNetv2-SSD and low-magnitude pruning concept as well as particular issues and how they were implemented in this work. Section 3 further provides evaluation metrics to evaluate the models' accuracy and performance. Next, section 4 explains post-training quantization and converting the model to TensorFlow lite format for mobile deployment. Section 5 points out the experimental setup and is followed by experimental results and analysis. The paper is concluded by section 6, containing the conclusion and future works.

2 PRUNING OBJECT DETECTION MODEL

2.1 Object detection model

There are multiple models for object detection tasks. They vary in layers architecture, number of parameters, and performance, typically divided into two-stage detectors which are usually slower but more accurate due to the fact that two-stage detectors involve two stages: region proposal and then the classification of those regions and refinement of the location prediction while single-stage detectors are faster but have less accuracy since they skip the region proposal stage and yields final localization and content prediction at once, giving them much faster speed than the former ones. Work by Syed et al. [31] provides an overview and comparison of different state-of-the-art object detection models, benchmarked by accuracy and frames per second that the model can run on. Some of the mentioned models in that paper suggest we can run on a real-time basis with certainly high accuracy, those would mentions: the two-stage detector YOLOv4 [2] which is more accurate but slower, single-stage detector MobileNetv2-SSD [5] which is faster but have less accuracy. In this work MobileNetv2-SSD is chosen for its better inference speed which is suitable for mobile devices but still maintains relatively good accuracy ([5], [31]).

2.2 Pruning on object detection models

Object detection models make use of convolutional neural networks (CNN) which have multiple expensive convolutional layers. Matrix multiplications are used to build the completely connected layers. The convolution layer convolves k x k kernels with n feature maps from preceding layer. If the following layer comprises m feature maps, then n x m convolutions and n x m x (H x W x k x k) multiply–accumulate operations are carried out, where W and H stand for the next layer's feature map's width and height. Thus it is important to reduce the complexity of the convolutional layers [1]. One way to achieve it is by using pruning techniques.

Pruning refers to the process of reducing the size of a neural network by removing unnecessary connections (weights) between neurons. The idea behind pruning is to identify and eliminate the connections that contribute less to the overall performance of the model. This helps to reduce the computational and memory requirements of the network, making it more efficient. Pruning can be traced back to early work by Le Cun et al. introducing Optimal Brain Damage, which reduces the number of parameters in a practical neural network [19]. Later work by Babak Hassibi and David G. Stork, Optimal Brain Surgeon further explore the possibilities of removing redundant parameters using second-order derivative information [14]. An illustration of pruning is given in Figure 1.

Several research studies pruning techniques on convolutional neural networks for object detection tasks ([24], [12], [13], [3], [22]) have been conducted over recent years. Recent research by Song Han et al. (2015) [13] claims to pruned state-of-the-art CNN models VGG16-SSD, reducing the number of parameters by 13x without loss in accuracy.

Pruning generally decreases the accuracy of the model as they remove parameters [32]. Thus pruning-fine tuning pipelines are created to minimize the accuracy loss. Fine-tuning involves training the pruned network on the original dataset or a subset of it. This step helps the network regain the performance lost during pruning and allows the remaining connections to adapt and relearn the important patterns in the data. This work applies the pipeline suggested by Liang Chen et al. [4] in their work, as shown in Figure 3.

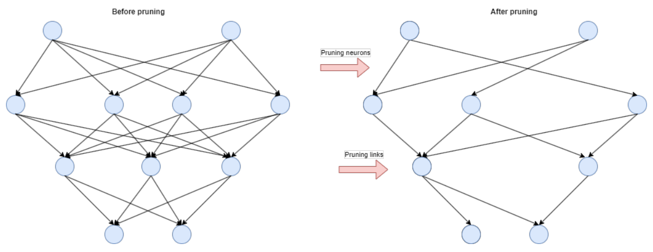

Figure 1: Pruning before and after

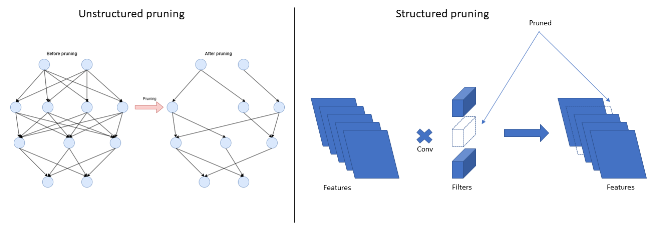

Figure 2: Schematic of two types of pruning methods.

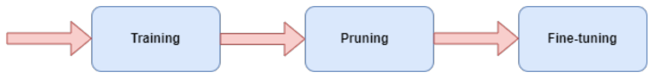

Figure 3: Typical pruning finetuning pipeline

There are two main categories of pruning: structure pruning, which refers to pruning the entire convolutional layers, channels, or filters, and unstructured pruning, which involves removing individual connections or weights in a neural network without any specific structure or pattern. An overview of the two categories of pruning can be seen in Figure 2 [4].

Another novel pruning technique could mention anchor pruning, which aims to reduce the number of anchor boxes used by the model to detect objects on an image, thus reducing the complexity of the network. Recent work from Maxim Bonnaerens et al. [3] well covered this method.

In this work, we focus on low-magnitude pruning, an unstructured pruning method that entails removing the insignificant weights, resulting in a more sparse network that is more suitable to deploy to Edge devices.

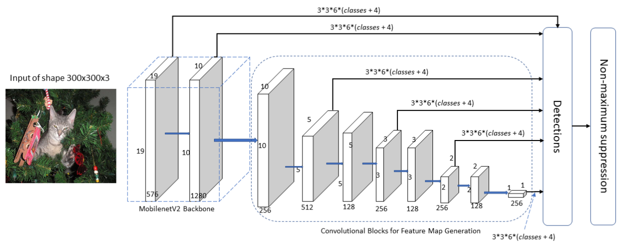

Figure 4: **SSD MobiletnetV2 Model Architecture, using a MobilenetV2 backbone.**

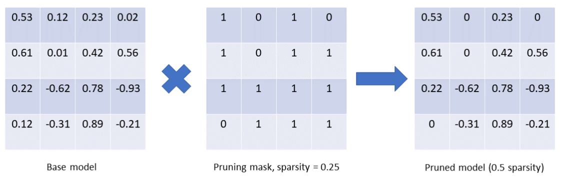

Figure 5: Illustration of how applying bit mask to zero out insignificant weights in a 4x4 weights matrix. The lowest 25% of weights are turned to 0.

3 PRUNING CANDIDATE SELECTION

3.1 SSD MobilenetV2

Single shot detector SSD MobilenetV2 was introduced by Yu-Chen Chiu et al. [5] in their work as an attempt to replace the traditional VGG16-SSD object detection model by replacing VGG16 backbone network with a less computationally expensive MobilenetV2 network. Figure 4 represents the architecture of the MobilenetV2-SSD model, with the original VGG16 backbone network replaced by a MobinetNetV2 backbone network. The backbone network is attached to the Single Shot Detector SSD head. The model generates six feature maps with different dimensions for the SSD detector head to perform object detection.

The entire implementation of this work is implemented on Jupyter Notebook hosted on Kaggle [18] for dedicated GPU for faster training process. SSD MobilenetV2 Object Detection model is implemented using Tensorflow APIs by attaching the supplied MobilenetV2 backbone network [29] with a Single Shot Detector head [21]. Resulting in a final model consisting of 178 layers with 8,527,790 parameters, which 8,493,678 of them are trainable parameters, allowing detection of up to a total of 200 objects belonging to 20 Pascal VOC classes. This model takes an image of shape 300x300x3 and result in 3 output float arrays representing the detected objects' labels, bounding boxes and confidence scores. The model was then initially trained on VOC 2007, VOC 2012 training datasets for 150 epochs with batch size of 32 and evaluated with VOC 2007 test dataset. The result acts as the base model for applying pruning techniques, which will be discussed next.

3.2 Low Magnitude Pruning

The most common and straightforward pruning technique is magnitude-based pruning, aims to remove weights or filters with small magnitudes in one or multiple layers, forcing them to become 0 until the layer(s) reach target sparsity levels, which is defined by:

$$sparsity\_level = \frac{parameters\_pruned}{total\_parameters}$$

Low-magnitude pruning can be done by having a bit mask that has the same size and shape as the layer, determining which weights to be zeroed out. The calculation of the bit mask for a weights matrix is as follows:

$$bit_i = 0 \text{ if } w_i \text{ is set to be pruned. Otherwise } bit_i = 1$$

Figure 5 further illustrate the ideas of low magnitude pruning by multiplying bit mask matrix with the weights matrix to zero out insignificant weights.

In order to find the appropriate amount of parameters to be pruned, in this work, different sparsity levels of global unstructured low magnitude pruning [8] is implemented using TensorFlow framework. Pruning will be applied to every Conv2D layer [7] of the MobileNetv2-SSD network, as they have the most parameters in the network thus it is possible that some of the convolutional layers are redundant. The parameters in the layers with prune low-magnitude applied on will continuously be pruned until those layers reach a level of sparsity. The higher sparsity level, the more parameters are pruned out of the model, giving faster inference speed but possibly lower the model's accuracy. After pruning, the model will be fine-tuned using the same dataset for another 20 epochs to improve the overall accuracy of the new pruned model by retraining unpruned parameters from their final values, following the pipeline as discussed in 2.2. Figure 6 visualize the weights pruning portion of layer "3_conv_boxes_output/Conv2D", a convolutional layer in the model, after being pruned with different levels of sparsity. Violet points represent weight being 0, while Yellow points represent weight different than 0. Since all 3 sparsity levels were pruned from a base model, the weights were pruned at sparsity level of 0.35 is included in the weights that were pruned in sparsity level of 0.5 and so on with sparsity level of 0.72.

3.3 Evaluation metrics

For object detection tasks, there are multiple metrics can be used to evaluate the models' accuracy, those could mention Intersec over Union (IoU) measuring the overlap between the predicted bounding boxes and the ground truth bounding boxes, Average Precision (AP) measuring the precision of object detection at various levels of recall, mean Average Precision (mAP) is the mean of average precisions across different object classes [25], etc. In this work,

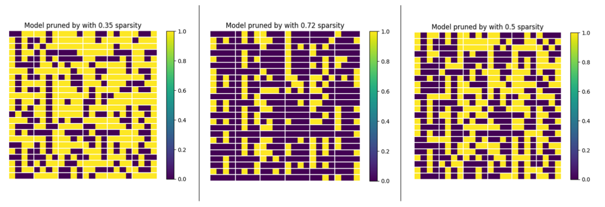

Figure 6: Visualization of a Convolutional Layer being pruned with 0.35, 0.72 and 0.5 sparsity with prune low-magnitude.

mAP is used for evaluating the models' accuracy as mAP evolves in the usage of both IoU and AP, providing an overall measure of the algorithm's performance by considering the performance of individual classes. The evaluation is performed over images data from the Pascal VOC test dataset with the computation of mAP implemented based on Pascal VOC Challenge article [10].
Additionally, the model size and frames per second rate of the Android application will also be measured to estimate the compression rate of the model as well as evaluate the precision-inference speed tradeoff of the new pruned models.

4 TENSORFLOW LITE AND POST TRAINING QUANTIZATION

TensorFlow lite is a set of tools that enable developers to execute models efficiently on mobile, embedded and edge devices with fast inference speed compared to normal TensorFlow models [9]. TensorFlow models can be converted into TensorFlow lite for mobile inference with options to quantize using TensorFlow's post-training quantization APIs, which was first introduced in June 2019 as an additional part for the TensorFlow's model optimization toolkit [30]. Quantization reduces the numerical precision of weights and activations thus reduces the memory and computational power required, improves latency performance for TensorFlow (lite) models while maintaining the model accuracy with minimal loss [16]. As a result of quantizing model with floating point representation of 32-bit data type to integer which is 8-bit data type, model's size can be 4x smaller and inference speed significantly improves. A study by Google Inc. team [16] shows that quantizated MobileNet model achieves up to 50% reduction in running time with a minimal loss in accuracy (around 1.8%), the test was made on a hardware-constrained mobile phone with Qualcomm Snapdragon 835 core.
In this work, the pruned models were converted to tensor flow lite format (tflite) using Tensorflow lite Converter APIs, allowing the model's execution on Android devices. The models were exported as three different formats default using float32 data type, quantized to float16 and quantized to integer8 data type. Due to the natural size of the data types, quantized float16 model will have 2x smaller and integer8 model will have 4x smaller size than the Float32 model, which makes the inference speed of those models faster.

5 EXPERIMENT AND RESULTS

The aim of the experiments is to first evaluate the pruning algorithms' effects on the models' size and accuracy and secondly assess the accuracy-speed tradeoffs on TensorFlow lite models over mobile devices to figure out at which sparsity levels and quantization methods the application can run with relatively high accuracy and speed.
In first step, pruning low-magnitude algorithm will be applied on MobileNetv2-SSD model with different levels of sparsity, namely 0, 0.35, 0.65, 0.72, 0.8. For each level of sparsity, mAP and model size will be calculated. mAP refers to the Precision of the model while lower model size can be helpful to lower the inference speed of the model.
Next, depending on the performance of pruned models, 2 pruned models will be selected to be further quantized and converted into TensorFlow lite format using TensorFlow converter APIs. Since TensorFlow lite converter supports post-training quantization to different data types, the quantized models will either be in default non-quantized float32 data type, quantized to float16 data type or in integer data type.
For each of the pruning-quantization method combinations, model size will be measured, mAP will be calculated based on VOC test data and FPS will be recorded while running an objects detection application on Samsung Galaxy S23 (and Google Glass Enterprise Edition 2) to analyze the trade-offs between model inference time and accuracy.

5.1 Dataset and data preprocessing

Object detection models often rely on huge amounts of data in form of images, with annotations of bounding boxes and labels for objects inside images. Some famous datasets for object detection tasks could be mentioned COCO, Pascal VOC, .. ([10], [20]). This work uses Pascal VOC 2007 and VOC 2012 object detection datasets for both training and validating as well as test data for evaluating the predictions for its availability over TensorFlow dataset loader. Pascal VOC consists of more than 20000 images of 20 classes. With that amount of data, object detection model should have enough input to be robust and with relatively high accuracy in detecting objects belonging to those 20 classes. In order to train the model, data needs to be preprocessed to match the input shape of the model, which is an image of size 300x300 with 3 color channels. Data will first be loaded using TensorFlow dataset loader, resized to match the model input size and convert to float32 data type. The final image data will be of shape (300, 300, 3) of data type float32, with each of the value between range [0, 1], matching the model's input layer's shape.

5.2 Devices

Both devices used for testing and evaluating the model use Android as their operating system. Thus, an Android Studio application is developed, allowing the evaluation of the model's performance visually by eyes as well as by measuring the frames per second rate. The application was made based on the sample application given by

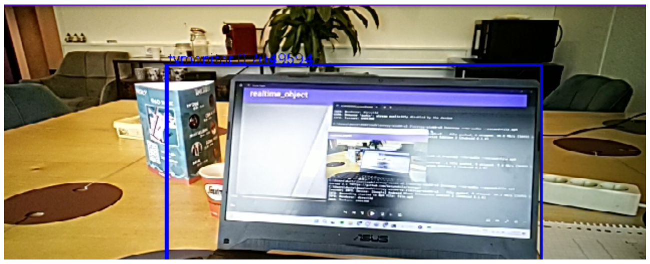

Figure 7: Example of the object detection app, laptop monitor is detected as class 'tvmonitor' with bounding box and probability displayed.

TensorFlow development team. The application itself was written in Kotlin as Kotlin having great supports from TensorFlow for importing the tflite model and executing tflite interference functions. TensorFlow lite model's execution graph will be loaded to memory, allowing the application to run inference on input image of shape 300x300x3 with data type of either float32 or int8. Since the input getting from devices' cameras is of a different resolution, additional input preprocessing steps need to be performed. The Android app continuously receives camera frames as bitmap input, converting them to TensorFlow buffers then resizes to 300x300x3 shape then feed the TensorFlow buffer to the model for inferencing. The data type of the buffer must be manually defined to match with the data type of our model, which is either float32 or int8 depends on the quantization method used. The output of the object detection model is 3 TensorFlow buffers representing labels, bounding boxes and confidence scores of the detected objects. Those TensorFlow buffers are visualized on the device's screen as blue boxes with the labels of the objects, confidence scores bounding boxes surrounding objects. An example is shown in Figure 7.

5.2.1 Samsung Galaxy S23. The application will first be deployed to Samsung Galaxy S23 using Qualcomm SM8550-AC Snapdragon 8 Gen 2 Octacore (1x3.36 GHz Cortex-X3 & 2x2.8 GHz Cortex-A715 & 2x2.8 GHz Cortex-A710 & 3x2.0 GHz Cortex-A510), Adreno 740 GPU and runs on Android 13. This is one of the most powerful commercial cellphones available in the market at the time of this work.

5.2.2 Google Glass Enterprise Edition 2. The application will also be installed on a Google Glass Enterprise Edition 2 for testing and evaluating the compressed models, comes with Octa-core Kryo (2 x 2.52 GHz, 6 x 1.7 GHz) CPU, Adreno 615 GPU, runs on Android 8.1. Google Glass Enterprise Edition 2 has lower computation power than Samsung Galaxy S23 due to weaker CPU and GPU. Thus giving us more input on the performance of the model over devices with fewer resources available.

5.3 Pruning results

First, Figure 8 shows the relation between levels of sparsity with the accuracy of the model and the model size. The horizontal line represents the mean Average Precision of the model, higher mAP means more accurate the model is. Vertical line meanwhile represents the size of the model measured in megabytes, lower model size suggests faster inference speed of the models. Each sparsity level is represented by a point in the figure, allowing a better overview of the size-accuracy trade-off.

Global sparsity level	0.00	0.35	0.5	0.65	0.72	0.80
mAP	0.489	0.508	0.52	0.479	0.472	0.429
Model size (MB)	31.83	23.77	19.54	15.11	12.84	10.22

Table 1: Pruning results.

Sparsity levels of 0.35 and 0.5 increase model accuracy and reduce the model size thus improving inference speed. Later levels of 0.65, 0.72 and 0.8 sparsity significantly reduce the model size however show a small drop in mAP of 0.01, 0.017, and 0.06 respectively compared to the base model. Pruning imposes tradeoffs between model efficiency and accuracy in those later sparsity levels while it is not the case on the former levels with sparsity levels equal or lower than 0.5. Thus, we can observe from pruning experiment that sparsity level does not always correlate with the accuracy and the size of the model. Indeed, we need to find out a sparsity level where the model gets the highest accuracy but with the most parameters pruned.

Among all sparsity levels in this work, the sparsity level of 0.5 gives the best accuracy (mAP = 0.52), which is even higher than the base non-pruned model (mAP of 0.52 versus mAP of 0.489 in the base model), with model size of 60% compared to the base non pruned model. It suggests that around 50% of the parameters in convolutional layers of MobileNetv2-SSD are actually helpful for the inference accuracy of the model, while the other parameters are redundant, leaving them non pruned cause the model to be less accurate. In this work, sparsity levels of 0.5 and 0.72 imposes good tradeoff between model size and mAP as sparsity level of 0.5 gives the highest accuracy with model size of $\tilde{6}1\%$ of the base model and sparsity level of 0.72 reduces the model size by 60% with minimal loss of 0.017 in mAP compared to the base model. Those two models will be further compressed using post-training quantization to perform experiments on Android devices.

5.4 Quantization and Android application performance

As in Table 2, quantization shows consistent results for the models with the same sparsity level in terms of accuracy, given that the model is converted to TensorFlow lite format. Despite a $\tilde{0}.060.07$ mAP lost in models' mAP when converting from TensorFlow model into TensorFlow lite format, the mean Average Precision of float16 and int8 models impose insignificant difference of 00.05 comparing to the float32 models, regardless the model is non pruned, pruned with sparsity level of 0.5 or pruned with sparsity level of 0.72. In all experimented models, float32 and float16 models' accuracy are roughly equal while int8 models show an insignificant loss in accuracy. Furthermore, table 2 shows that integer quantized models show a significant increase in inference speed, represented by frames per second measured on Samsung Galaxy S23. Non-pruned

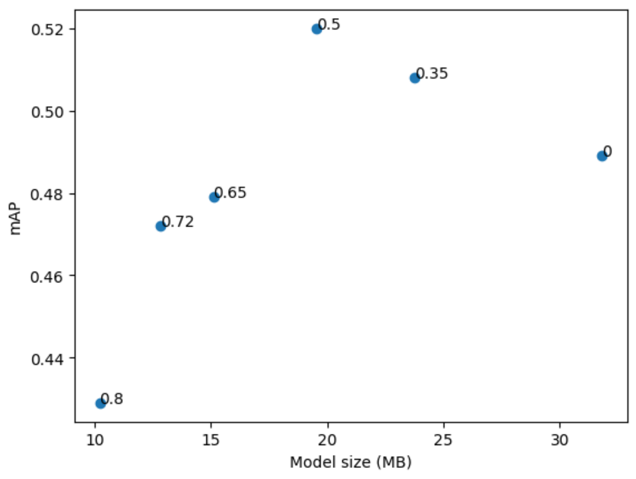

Figure 8: mAP and Model size with different levels of sparsity

Model	mAP	Model size	FPS (SS S23)	FPS (Glass EE2)
model_nonprune_base	0.489	31.83		
model_nonprune_float32_tflite	0.424	31.61	11	7.57
model_nonprune_float16_tflite	0.426	15.82	12	7.57
model_nonprune_int8_tflite	0.421	6.7	15	10.41
model_prune50_base	0.52	19.54		
model_prune50_float32_tflite	0.465	19.44	13	7.64
model_prune50_float16_tflite	0.465	10.77	14	7.69
model_prune50_int8_tflite	0.461	5.18	16	10.54
model_prune72_base	0.472	12.84		
model_prune72_float32_tflite	0.409	12.77	14	7.81
model_prune72_float16_tflite	0.409	7.5	15	7.91
model_prune72_int8_tflite	0.404	3.66	18	10.72

Table 2: Models accuracy, size and performance for different sparsity levels and quantization data types.

model is speeded up by 36% when quantized to integer, while this number is 23% and 28% for models with sparsity levels of 0.5 and 0.72 respectively. Meanwhile, TensorFlow lite models in float16 data type show small amount of frames per second rate improvement of 1 comparing to the base model, regardless the pruning amount either be 0, 0.5 or 0.72 sparsity. Thus, integer quantization of MobileNetv2-SSD model should be performed over pruned models to first compress the size of the model and second improve the inference speed with insignificant loss in accuracy.

While pruning and quantization show small losses in mAP, the TensorFlow lite models' sizes are exponentially reduced depending on the compression techniques used. The more parameters pruned or the more compact data types used, the smaller model size will be. In this work, the model with sparsity level of 0.72 and quantized to integers have a compression rate of 9x smaller than the base, non pruned model with a minimal 0.08 mAP reduction in terms of accuracy.

Figure 9 shows the correlation between model size and speed in form of frames per second. The horizontal axis represents the TensorFlow lite models' size measured in megabytes while the vertical axis measures the frames per second rate of those models. Three sparsity levels are displayed in this figure: 0, 0.5, 0.72.

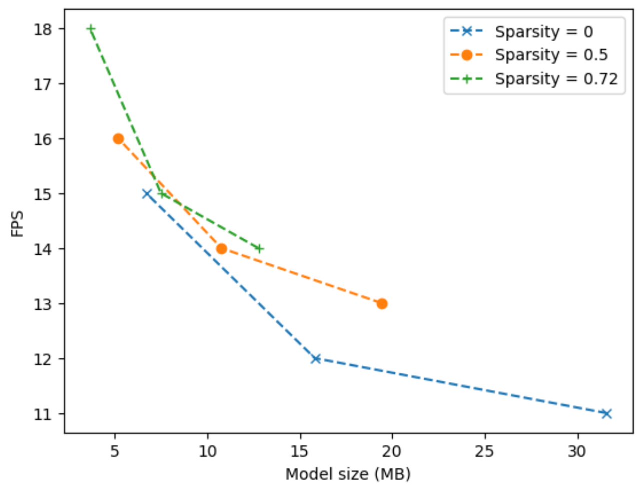

Figure 9: Correlations between model size and FPS

Following Figure 9, the frames per second rates of object detection application, which represent the inference speed of the (compressed) model(s), are correlative with the size of the model. The lower the model size is, the faster inference speed the model can run. Sparsity level of 0 without quantization gives the worst FPS performance and sparsity level of 0.72 with integer quantization gives the best FPS performance due to the fact that the latter model is 9x smaller in size than the former. Furthermore, in this work, integer quantized models have advantages over others float models in terms of frames per second rate for about 3-4 frames per second on a Samsung Galaxy S23 with minimal trade-offs in accuracy. Table 2 shows the same trends for FPS when deployed on Google Glass Enterprise Edition 2. Hence it suggests that object detection models can retain relatively good accuracy when quantizing models to different data types namely float32, float16, int8. Thus, developers can safely quantize TensorFlow lite models to Integers to improve models' performance with minimal loss in accuracy.

In this work, by combining pruning with sparsity level of 0.72 and quantization to integer data type, the new TensorFlow lite model can be speeded up by 7 FPS which is 60% improvement comparing to non pruned model in float32 data format, while reducing accuracy by 4% (0.424 to 0.404). Meanwhile, pruning with sparsity level of 0.5 and quantization to integer data type gives improvements of 45% speed up and increase accuracy by 8%. Either way impose a good trade-off between accuracy and inference speed. In general, the higher level of sparsity, the faster the model is. Depending on the requirements of the applications regarding performance and accuracy, developers can thus choose the best sparsity level to match accuracy and speed criteria before converting the model to integer TensorFlow lite model.

We further deployed the model to Google Glass Enterprise Edition 2 to evaluate the performance on a standalone hardware-constrained small device. As shown in Table 2, all float models can run with over 7 FPS, integer quantized models can run with 10+ FPS. This indeed suggests that we can replace traditional object detection applications on edge devices, which rely on streaming the recorded input to a powerful server side, predicting the output and returning the output back to the application side, with object detection applications using pruned and quantized TensorFlow lite models. This way, the performance of the application not be affected by the network connectivity, an unpredictable variable and not always available. With a network of average 50 milliseconds one-way delay, the streaming-based applications can only have a maximum of 10 FPS due to two-way data traffic delay, which is lower than the application using pruned and integer quantized TensorFlow Lite models.

6 CONCLUSION AND FUTUREWORK

In this work, we pruned state of the art object detection model MobileNetv2-SSD with different sparsity levels and quantized using TensorFlow APIs for better performance on hardware-constrained devices. For pruning, it is important to find the suitable sparsity level that imposes reasonable trade-offs between accuracy and performance that match developers' needs. For quantization, models can be safely converted to TensorFlow lite format using TensorFlow lite converter and quantized to smaller data types to compress the model by two times using float16 data type or four times using 8-bit integer8 data type. Using both methods, the model can be compressed up to 9x smaller, giving around 60% improvement in frames per second performance on Android devices with minimal loss in accuracy. Moreover, the new model is deployed on hardware-constrained Google Glass Enterprise Edition 2 giving interesting results of 10+ frames per second.
This work can be extended to experiment with different model(s), for instance, YOLOv5, and with different prune method(s), such as structural pruning and pruning in specific layer(s) as for each model, there would possibly be different sparsity level(s) as well as pruning method(s) giving best result on accuracy-performance tradeoff.

REFERENCES

[1] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. 2017. Structured pruning of deep convolutional neural networks. *ACM Journal on Emerging Technologies in Computing Systems*, 13, (July 2017), 1–18, 3, (July 2017). DOI: 10.1145/3005348.

[2] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. 2020. Yolov4: optimal speed and accuracy of object detection. *CoRR*, abs/2004.10934. https://arxiv.org/abs/2004.10934.

[3] Maxim Bonnaerens, Matthias Freiberger, and Joni Dambre. 2022. Anchor pruning for object detection. *Computer Vision and Image Understanding*, 221, (Aug. 2022), 103445. DOI: 10.1016/j.cviu.2022.103445.

[4] Liyang Chen, Yongquan Chen, Juntong Xi, and Xinyi Le. 2022. Knowledge from the original network: restore a better pruned network with knowledge distillation. *Complex Intelligent Systems*, 8, (Apr. 2022), 709–718, 2, (Apr. 2022). DOI: 10.1007/s40747-020-00248-y.

[5] Yu-Chen Chiu, Chi-Yi Tsai, Mind-Da Ruan, Guan-Yu Shen, and Tsu-Tian Lee. 2020. Mobilenet-ssdv2: an improved object detection model for embedded systems. In IEEE, (Aug. 2020), 1–5. ISBN: 978-1-7281-5960-7. DOI: 10.1109/ICSSE50014.2020.9219319.

[6] Tejalal Choudhary, Vipul Mishra, Anurag Goswami, and Jagannathan Sarangapani. 2020. A comprehensive survey on model compression and acceleration. *Artificial Intelligence Review*, 53, 5113–5155, 7. DOI: 10.1007/s10462-020-09816-7.

[7] TensorFlow Developers. [n. d.] Conv2d. https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2D.

[8] TensorFlow Developers. [n. d.] Prune low magnitude. https://www.tensorflow.org/model_optimization/api_docs/python/tfmot/sparsity/keras/prune_low_magnitude.

[9] TensorFlow Developers. [n. d.] Tensorflow lite overview. https://www.tensorflow.org/lite/guide.

[10] Mark Everingham, Luc Van Gool, Christopher K. I. Williams, John Winn, and Andrew Zisserman. 2010. The pascal visual object classes (voc) challenge. *International Journal of Computer Vision*, 88, (June 2010), 303–338, 2, (June 2010). DOI: 10.1007/s11263-009-0275-4.

[11] Hugging Face. [n. d.] Quantization. https://huggingface.co/docs/optimum/concept_guides/quantization.

[12] Song Han, Huizi Mao, and William J Dally. 2016. Deep compression: compressing deep neural networks with pruning, trained quantization and huffman coding. (2016).

[13] Song Han, Jeff Pool, John Tran, and William J Dally. 2015. Learning both weights and connections for efficient neural networks. *CoRR*, abs/1506.02626. http://arxiv.org/abs/1506.02626.

[14] Babak Hassibi and David G.Stork. 1992. Second order derivatives for network pruning: optimal brain surgeon. *Adv Neural Inform Proc Syst*, 5, (May 1992).

[15] Zejiang Hou et al. 2022. Chex: channel exploration for cnn model compression. (2022).

[16] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew G Howard, Hartwig Adam, and Dmitry Kalenichenko. 2017. Quantization and training of neural networks for efficient integer-arithmetic-only inference. *CoRR*, abs/1712.05877. http://arxiv.org/abs/1712.05877.

[17] Sambhav R Jain, Albert Gural, Michael Wu, and Chris Dick. 2019. Trained uniform quantization for accurate and efficient neural network inference on fixed-point hardware. *CoRR*, abs/1903.08066. http://arxiv.org/abs/1903.08066.

[18] Kaggle. [n. d.] Kaggle. https://www.kaggle.com/.

[19] Yann Lecun, John Denker, and Sara Solla. 1989. Optimal brain damage. In vol. 2. (May 1989), 598–605.

[20] Tsung-Yi Lin et al. 2014. Microsoft coco: common objects in context. *CoRR*, abs/1405.0312. http://arxiv.org/abs/1405.0312.

[21] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. 2015. Ssd: single shot multibox detector, (Dec. 2015). DOI: 10.1007/978-3-319-46448-0_2.

[22] Zonglei Lyu, Dan Zhang, and Jia Luo. 2022. A gpu-free real-time object detection method for apron surveillance video based on quantized mobilenet-ssd. *IET Image Processing*, 16, (June 2022), 2196–2209, 8, (June 2022). DOI: 10.1049/ipr2.12483.

[23] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. 2016. Pruning convolutional neural networks for resource efficient transfer learning. *CoRR*, abs/1611.06440. http://arxiv.org/abs/1611.06440.

[24] Gedeon Muhawenayo and Georgia Gkioxari. 2021. Compressed object detection. *CoRR*, abs/2102.02896. https://arxiv.org/abs/2102.02896.

[25] Rafael Padilla, Sergio L. Netto, and Eduardo A. B. da Silva. 2020. A survey on performance metrics for object-detection algorithms. In IEEE, (July 2020), 237–242. ISBN: 978-1-7281-7539-3. DOI: 10.1109/IWSSIP48289.2020.9145130.

[26] Morteza Mousa Pasandi, Mohsen Hajabdollahi, Nader Karimi, and Shadrokh Samavi. 2020. Modeling of pruning techniques for deep neural networks simplification. *CoRR*, abs/2001.04062. https://arxiv.org/abs/2001.04062.

[27] David A Patterson, Joseph Gonzalez, Quoc V Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David R So, Maud Texier, and Jeff Dean. 2021. Carbon emissions and large neural network training. *CoRR*, abs/2104.10350. https://arxiv.org/abs/2104.10350.

[28] Haotong Qin, Ruihao Gong, Xianglong Liu, Xiao Bai, Jingkuan Song, and Nicu Sebe. 2020. Binary neural networks: a survey. *CoRR*, abs/2004.03333. https://arxiv.org/abs/2004.03333.

[29] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Mobilenetv2: inverted residuals and linear bottlenecks, (Jan. 2018).

[30] TensorFlow Model Optimization Team. [n. d.] Tensorflow model optimization toolkit — post-training integer quantization. https://blog.tensorflow.org/2019/06/tensorflow-integer-quantization.html.

[31] Syed Sahil Abbas Zaidi, Mohammad Samar Ansari, Asra Aslam, Nadia Kanwal, Mamoona Asghar, and Brian Lee. 2022. A survey of modern deep learning based object detection models. *Digital Signal Processing*, 126, (June 2022), 103514. DOI: 10.1016/j.dsp.2022.103514.

[32] Michael Zhu and Suyog Gupta. 2017. To prune, or not to prune: exploring the efficacy of pruning for model compression, (Oct. 2017).